# finufft Documentation

***Release 1.0***

**Alex Barnett and Jeremy Magland**

August 20, 2018

**1 Installation** **3**
1.1 Obtaining FINUFFT . . . 3
1.2 Dependencies . . . 3
1.3 Compilation . . . 4
1.4 Building the python wrappers . . . 5

**2 Mathematical definitions of transforms** **7**

**3 Contents of the package** **9**

**4 Usage and interfaces** **11**
4.1 Interfaces from C++ . . . 11
4.2 Interfaces from C . . . 19
4.3 Interfaces from fortran . . . 19
4.4 Design notes and data types . . . 20

**5 Advanced interfaces for many vectors with same nonuniform points** **21**
5.1 2D transforms . . . 21
5.2 Design notes . . . 22

**6 MATLAB/octave interfaces** **23**

**7 Python interface** **31**

**8 Julia interface** **43**

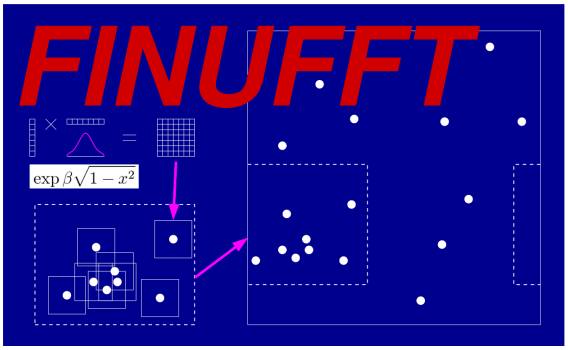
**9 Related packages** **45**
9.1 Interfaces to FINUFFT from other languages . . . 45
9.2 Packages making use of FINUFFT . . . 45

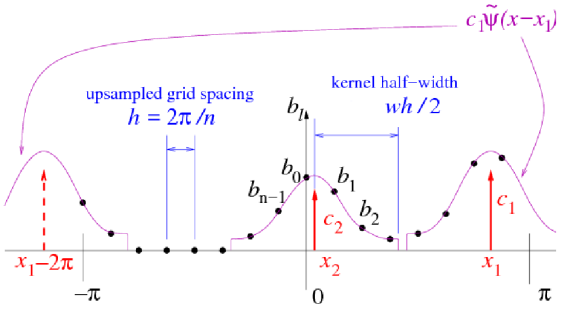
**10 Known Issues** **47**
10.1 Issues with library . . . 47
10.2 Issues with interfaces . . . 47
10.3 Bug reports . . . 47

**11 Acknowledgments** **49**

**12 References** **51**

**Index** **53**




FINUFFT is a set of libraries to compute efficiently three types of nonuniform fast Fourier transform (NUFFT) to a specified precision, in one, two, or three dimensions, on a multi-core shared-memory machine. The library has a very simple interface, does not need any precomputation step, is written in C++ (using OpenMP and FFTW), and has wrappers to C, fortran, MATLAB, octave, and python. As an example, given $M$ arbitrary real numbers $x_j$ and complex numbers $c_j$, with $j = 1, \ldots, M$, and a requested integer number of modes $N$, the 1D type-1 (aka "adjoint") transform evaluates the $N$ numbers

$$f_k = \sum_{j=1}^{M} c_j e^{ikx_j} \ , \qquad \text{for } \ k \in \mathbb{Z}, \quad -N/2 \le k \le N/2 - 1 \ . \tag{1}$$

The $x_j$ can be interpreted as nonuniform source locations, $c_j$ as source strengths, and $f_k$ then as the $k$th Fourier series coefficient of the distribution $f(x) = \sum_{j=1}^{M} c_j \delta(x - x_j)$. Such exponential sums are needed in many applications in science and engineering, including signal processing, imaging, diffraction, and numerical partial differential equations. The naive CPU effort to evaluate (1) is $O(NM)$. The library approximates (1) to a requested relative precision $\epsilon$ with nearly linear effort $O(M \log(1/\epsilon) + N \log N)$. Thus the speedup over the naive cost is similar to that achieved by the FFT. This is achieved by spreading onto a regular grid using a carefully chosen kernel, followed by an upsampled FFT, then a division (deconvolution) step. For the 2D and 3D definitions, and other types of transform, see below.

The FINUFFT library achieves its speed via several innovations including:

1. The use of a new spreading kernel that is provably close to optimal, yet faster to evaluate than the Kaiser-Bessel kernel
2. Quadrature approximation for the Fourier transform of the spreading kernel
3. Load-balanced multithreading of the type-1 spreading operation

For the same accuracy in 3D, the library is 3-50 times faster on a single core than the single-threaded fast Gaussian gridding CMCL libraries of Greengard-Lee, and in the multi-core setting for spreading-dominated problems is faster than the Chemnitz NFFT3 library even when the latter is allowed a RAM-intensive full precomputation of the kernel. This is especially true for highly non-uniform point distributions and/or high precision. Our library does not require precomputation, and uses minimal RAM.

For the case of small problems where repeated NUFFTs are needed with a fixed set of nonuniform points, we have started to build advanced interfaces for this case. These are a factor of 2 or more faster than repeated calls to the plain interface, since certain costs such as FFTW setup and sorting are performed only once.

**Note:** For very small repeated problems (less than 10000 input and output points), users should also consider a dense matrix-matrix multiplication against the NUDFT matrix using BLAS3 (eg ZGEMM). Since we did not want BLAS to be a dependency, we have not yet included this option.

CHAPTER

ONE

# INSTALLATION

## Obtaining FINUFFT

Go to the github page https://github.com/ahbarnett/finufft and follow instructions (eg see the green button).

## Dependencies

This library is currently supported for unix/linux and also tested on Mac OSX. We have heard that it can be compiled on Windows too.

For the basic libraries

- C++ compiler, such as `g++` packaged with GCC
- FFTW3
- GNU make

Optional:

- `numdiff` (preferred but not essential; enables pass-fail math validation)
- for Fortran wrappers: compiler such as `gfortran`
- for matlab/octave wrappers: MATLAB, or octave and its development libraries
- for building new matlab/octave wrappers (experts only): `mwrap`
- for the python wrappers you will need `python` and `pip` (if you prefer python v2), or `python3` and `pip3` (for python v3). You will also need `pybind11`

### Tips for installing dependencies on various operating systems

On a Fedora/CentOS linux system, these dependencies can be installed as follows:

```
sudo yum install make gcc gcc-c++ gcc-gfortran fftw3 fftw3-devel libgomp octave octave-devel
```

then see below for `numdiff` and `mwrap`.

**Note:** we are not exactly sure how to install python3 and pip3 using yum

then download the latest `numdiff` from http://gnu.mirrors.pair.com/savannah/savannah/numdiff/ and set it up via `./configure; make; sudo make install`

On Ubuntu linux (assuming python3 as opposed to python):

```
sudo apt-get install make build-essential libfftw3-dev gfortran numdiff python3 python3-pip octave li
```

On Mac OSX:

Make sure you have `make` installed, eg via XCode.

Install gcc, for instance using pre-compiled binaries from http://hpc.sourceforge.net/

Install homebrew from http://brew.sh:

```
brew install fftw
```

Install `numdiff` as below.

(Note: we are not exactly sure how to install python3 and pip3 on mac)

Currently in Mac OSX, `make lib` fails to make the shared object library (.so); however the static (.a) library is of reasonable size and works fine.

### Installing numdiff

numdiff by Ivano Primi extends `diff` to assess errors in floating-point outputs. Download the latest `numdiff` from the above URL, un-tar the package, cd into it, then build via `./configure; make; sudo make install`

### Installing MWrap

This is not needed for most users. MWrap is a very useful MEX interface generator by Dave Bindel. Make sure you have `flex` and `bison` installed. Download version 0.33 or later from http://www.cs.cornell.edu/~bindel/sw/mwrap, un-tar the package, cd into it, then:

```
make
sudo cp mwrap /usr/local/bin/
```

## Compilation

We first describe compilation for default options (double precision, openmp) via GCC. If you have a nonstandard unix environment (eg a Mac) or want to change the compiler, then place your compiler and linking options in a new file `make.inc`. For example such files see `make.inc.*`. See `makefile` for what can be overridden.

Compile and do a rapid (less than 1-second) test of FINUFFT via:

```
make test
```

This should compile the main libraries then run tests which should report zero crashes and zero fails. (If numdiff was not installed, it instead produces output that you will have to check by eye matches the requested accuracy.)

Use `make perftest` for larger spreader and NUFFT tests taking 15-30 seconds.

Run `make` without arguments for full list of possible make tasks.

Note that the library includes the C and fortran interfaces defined in `src/finufft_c.h` and `fortran/finufft_f.h` respectively. If there is an error in testing on a standard set-up, please file a bug report as a New Issue at https://github.com/ahbarnett/finufft/issues

### Custom library compilation options

You may want to make the library for other data types. Currently library names are distinct for single precision (libfinufftf) vs double (libfinufft). However, single-threaded vs multithreaded are built with the same name, so you will have to move them to other locations, or build a 2nd copy of the repo, if you want to keep both versions.

You *must* do at least `make objclean` before changing precision or openmp options.

**Single precision**: append `PREC=SINGLE` to the make task. Single-precision saves half the RAM, and increases speed slightly (<20%). The C++, C, and fortran demos are all tested in single precision. However, it will break matlab, octave, python interfaces.

**Single-threaded**: append `OMP=OFF` to the make task.

### Building examples and wrappers

`make examples` to compile and run the examples for calling from C++ and from C.

The `examples` and `test` directories are good places to see usage examples.

`make fortran` to compile and run the fortran wrappers and examples.

`make matlab` to build the MEX interface to matlab.

`make octave` to build the MEX-like interface to octave.

On Mac OSX, we have found that the MATLAB MEX settings need to be overridden: edit the file `mex_C++_maci64.xml` in the MATLAB distro, to read, for instance:

```
CC="gcc-8"
CXX="g++-8"
CFLAGS="-ansi -D_GNU_SOURCE -fexceptions -fPIC -fno-omit-frame-pointer -pthread"
CXXFLAGS="-ansi -D_GNU_SOURCE -fPIC -fno-omit-frame-pointer -pthread"
```

These settings are copied from the `glnxa64` case. Here you will want to replace the compilers by whatever version of GCC you have installed. For pre-2016 MATLAB Mac OSX versions you'll instead want to edit the `maci64` section of `mexopts.sh`.

## Building the python wrappers

First make sure you have python3 and pip3 (or python and pip) installed and that you have already compiled the C++ library (eg via `make lib`). Python links to this compiled library. Next make sure you have NumPy and pybind11 installed:

```
pip3 install numpy pybind11
```

You may then do `make python3` which calls pip3 for the install then runs some tests. An additional test you could do is:

```
python3 run_speed_tests.py
```

In all the above the "3" can be omitted if you want to work with python v2.

See also Dan Foreman-Mackey's earlier repo that also wraps finufft, and from which we have drawn code: python-finufft

## A few words about python environments

There can be confusion and conflicts between various versions of python and installed packages. It is therefore a very good idea to use virtual environments. Here's a simple way to do it (after installing python-virtualenv):

```
Open a terminal
virtualenv -p /usr/bin/python3 env1
. env1/bin/activate
```

Now you are in a virtual environment that starts from scratch. All pip installed packages will go inside the env1 directory. (You can get out of the environment by typing `deactivate`)

CHAPTER

TWO

# MATHEMATICAL DEFINITIONS OF TRANSFORMS

We use notation with a general space dimensionality $d$, which will be 1, 2, or 3, in our library. The arbitrary (ie nonuniform) points in space are denoted $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \dots, M$. We will see that for type-1 and type-2, without loss of generality one could restrict to the periodic box $[-\pi, \pi)^d$. For type-1 and type-3, each such NU point carries a given associated strength $c_j \in \mathbb{C}$. Type-1 and type-2 involve the Fourier "modes" (Fourier series coefficients) with integer indices lying in the set

$$K = K_{N_1,\dots,N_d} := K_{N_1} K_{N_2} \dots K_{N_d} \ ,$$

where

$$K_{N_i} := \begin{cases} \{-N_i/2, \dots, N_i/2 - 1\}, & N_i \text{ even}, \\ \{-(N_i-1)/2, \dots, (N_i-1)/2\}, & N_i \text{ odd}. \end{cases}$$

For instance, $K_{10} = \{-5, -4, \dots, 4\}$, whereas $K_{11} = \{-5, -4, \dots, 5\}$. Thus, in the 1D case $K$ is an interval containing $N_1$ integer indices, in 2D it is a rectangle of $N_1 N_2$ index pairs, and in 3D it is a cuboid of $N_1 N_2 N_3$ index triplets.

Then the type-1 (nonuniform to uniform, aka "adjoint") NUFFT evaluates

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{k} \cdot \mathbf{x}_j} \qquad \text{for } \mathbf{k} \in K \tag{2.1}$$

This can be viewed as evaluating a set of Fourier series coefficients due to sources with strengths $c_j$ at the arbitrary locations $\mathbf{x}_j$. Either sign of the imaginary unit in the exponential can be chosen in the interface. Note that our normalization differs from that of references [DR,GL].

The type-2 (U to NU, aka "forward") NUFFT evaluates

$$c_j := \sum_{\mathbf{k} \in K} f_{\mathbf{k}} e^{\pm i \mathbf{k} \cdot \mathbf{x}_j} \qquad \text{for } j = 1, \dots, M \tag{2.2}$$

This is the adjoint of the type-1, ie the evaluation of a given Fourier series at a set of arbitrary points. Both type-1 and type-2 transforms are invariant under translations of the NU points by multiples of $2\pi$, thus one could require that all NU points live in the origin-centered box $[-\pi, \pi)^d$. In fact, as a compromise between library speed, and flexibility for the user (for instance, to avoid boundary points being flagged as outside of this box due to round-off error), our library only requires that the NU points lie in the three-times-bigger box $\mathbf{x}_j \in [-3\pi, 3\pi]^d$. This allows the user to choose a convenient periodic domain that does not touch this three-times-bigger box. However, there may be a slight speed increase if most points fall in $[-\pi, \pi)^d$.

Finally, the type-3 (NU to NU) transform does not have restrictions on the NU points, and there is no periodicity. Let $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \dots, M$, be NU locations, with strengths $c_j \in \mathbb{C}$, and let $\mathbf{s}_k$, $k = 1, \dots, N$ be NU frequencies. Then the type-3 transform evaluates:

$$f_{\mathbf{k}} := \sum_{j=1}^{M} c_j e^{\pm i \mathbf{s}_k \cdot \mathbf{x}_j} \qquad \text{for } k = 1, \dots, N \tag{2.3}$$

For all three transforms, the computational effort scales like the product of the space-bandwidth products (real-space width times frequency-space width) in each dimension. For type-1 and type-2 this means near-linear scaling in the total number of modes $N := N_1 \dots N_d$. However, be warned that for type-3 this means that, even if $N$ and $M$ are small, if the product of the tightest intervals enclosing the coordinates of $\mathbf{x}_j$ and $\mathbf{s}_k$ is large, the algorithm will be inefficient. For such NU points, a direct sum should be used instead.

We emphasise that the NUFFT tasks that this library performs should not be confused with either the discrete Fourier transform (DFT), the (continuous) Fourier transform (although it may be used to approximate this via a quadrature rule), or the inverse NUFFT (the iterative solution of the linear system arising from nonuniform Fourier sampling, as in, eg, MRI). It is also important to know that, for NU points, *the type-1 is not the inverse of the type-2*. See the references for clarification.

CHAPTER

THREE

# CONTENTS OF THE PACKAGE

- `finufft-manual.pdf` : the manual (auto-generated by sphinx)
- `docs` : source files for documentation (.rst files are human-readable)
- `README.md` : github-facing (and human text-reader) doc info
- `LICENSE` : how you may use this software
- `CHANGELOG` : list of changes, release notes
- `TODO` : list of things needed to fix or extend (hackers please help)
- `makefile` : GNU makefile (there are no makefiles in subdirectories)
- `src` : main library source and headers. Compiled objects will be built here
- `lib` : dynamic library will be built here
- `lib-static` : static library will be built here
- `test` : validation and performance tests, bash scripts driving compiled C++
  - `test/check_finufft.sh` is the main pass-fail validation bash script
  - `test/nuffttestnd.sh` is a simple uniform-point performance test bash script
  - `test/results` : validation comparison outputs (*.refout; do not remove these), and local test and performance outputs (*.out; you may remove these)
- `examples` : simple example codes for calling the library from C++ and C
- `fortran` : wrappers and drivers for Fortran (see `fortran/README`)
- `matlab` : wrappers and examples for MATLAB/octave
- `finufftpy` : python wrappers
- `python_tests` : accuracy and speed tests and examples using the python wrappers
- `setup.py` : needed so pip or pip3 can build and install the python wrappers
- `contrib` : 3rd-party code

CHAPTER

FOUR

# USAGE AND INTERFACES

Here we describe calling FINUFFT from C++, C, and Fortran.

We provide Type 1 (nonuniform to uniform), Type 2 (uniform to nonuniform), and Type 3 (nonuniform to nonuniform), in dimensions 1, 2, and 3. This gives nine basic routines. There are also two *advanced interfaces* for multiple 2d1 and 2d2 transforms with the same point locations.

Using the library is a matter of filling your input arrays, allocating the correct output array size, possibly setting fields in the options struct, then calling one of the transform routines below.

## Interfaces from C++

We first give a simple example of performing a 1D type-1 transform in double precision from C++, the library's native language, using C++ complex number type. First include the headers:

```
#include "finufft.h"
#include <complex>
using namespace std;
```

Now in the body of the code, assuming `M` has been set to be the number of nonuniform points, we allocate the input arrays:

```
double *x = (double *)malloc(sizeof(double)*M);
complex<double>* c = (complex<double>*)malloc(sizeof(complex<double>)*M);
```

These arrays should now be filled with the user's data: values in `x` should lie in $[-3\pi, 3\pi]$, and `c` can be arbitrary complex strengths (we omit example code for this here). With `N` as the number of modes, allocate the output array:

```
complex<double>* F = (complex<double>*)malloc(sizeof(complex<double>)*N);
```

Before usage, set default values in the options struct `opts`:

```
nufft_opts opts; finufft_default_opts(opts);
```

Warning: if this is not called, options may take on random values which may cause a crash. To perform the nonuniform FFT is then one line:

```
int ier = finufft1d1(M,x,c,+1,1e-6,N,F,opts);
```

This fills `F` with the output modes, in increasing ordering from `-N/2` to `N/2-1`. Here `+1` sets the sign of `i` in the exponentials in the *definitions*, `1e-6` chooses 6-digit relative tolerance, and `ier` is a status output which is zero if successful (see below). See `example1d1.cpp`, in the `examples` directory, for a simple full working example. Then to compile, linking to the double-precision static library, use eg:

```
g++ example1d1.cpp -o example1d1 -I FINUFFT/src FINUFFT/lib-static/libfinufft.a -fopenmp -lfftw3_thre
```

where `FINUFFT` denotes the top-level directory of the installed library. The `examples` and `test` directories are good places to see further usage examples. The documentation for all nine routines follows below.

If you have a small-scale 2D task (say less than $10^5$ points or modes) with multiple strength or coefficient vectors but fixed nonuniform points, see the *advanced interfaces*.

## Options

You may override the default options in `opts` by changing the fields in this struct, after setting up with default values as above. This allows control of various parameters such as the mode ordering, FFTW plan mode, upsampling factor $\sigma$, and debug/timing output. Here is the list of the options fields you may set (see the header `src/finufft.h`). Here the abbreviation `FLT` means `double` if compiled in the default double-precision, or `single` if single precision:

```
int debug;          // 0: silent, 1: text basic timing output
int spread_debug;   // passed to spread_opts, 0 (no text) 1 (some) or 2 (lots)
int spread_sort;    // passed to spread_opts, 0 (don't sort) 1 (do) or 2 (heuristic)
int spread_kerevalmeth; // "     spread_opts, 0: exp(sqrt()), 1: Horner ppval (faster)
int spread_kerpad;  // passed to spread_opts, 0: don't pad to mult of 4, 1: do
int chkbnds;        // 0: don't check if input NU pts in [-3pi,3pi], 1: do
int fftw;           // 0:FFTW_ESTIMATE, or 1:FFTW_MEASURE (slow plan but faster)
int modeord;        // 0: CMCL-style increasing mode ordering (neg to pos), or
                    // 1: FFT-style mode ordering (affects type-1,2 only)
FLT upsampfac;      // upsampling ratio sigma, either 2.0 (standard) or 1.25 (small FFT)
```

Here are their default settings (set in `src/common.cpp:finufft_default_opts`):

```
debug = 0;
spread_debug = 0;
spread_sort = 2;
spread_kerevalmeth = 1;
spread_kerpad = 1;
chkbnds = 0;
fftw = FFTW_ESTIMATE;
modeord = 0;
upsampfac = (FLT)2.0;
```

To get the fastest runtime, we recommend that you experiment firstly with: `fftw`, `upsampfac`, and `spread_sort`, detailed below. If you are having crashes, set `chkbnds=1` to see if illegal `x` coordinates are being input.

Notes on various options:

`spread_sort`: the default setting is `spread_sort=2` which applies the following heuristic rule: in 2D or 3D always sort, but in 1D, only sort if N (number of modes) > M/10 (where M is number of nonuniform pts).

`fftw`: The default FFTW plan is `FFTW_ESTIMATE`; however if you will be making multiple calls, consider `fftw=FFTW_MEASURE`, which will spend many seconds planning but give the fastest speed when called again. Note that FFTW plans are saved (by FFTW's library) automatically from call to call in the same executable (incidentally also in the same MATLAB/octave or python session).

`upsampfac`: This is the internal factor by which the FFT is larger than the number of requested modes in each dimension. We have built efficient kernels for only two settings: `upsampfac=2.0` (standard), and `upsampfac=1.25` (lower RAM, smaller FFTs, but wider spreading kernel). The latter can be much faster when the number of nonuniform points is similar or smaller to the number of modes, and/or if low accuracy is required. It is especially much faster for type 3 transforms. However, the kernel widths $w$ are about 50% larger in each dimension, which can lead to slower spreading (it can also be faster due to the smaller size of the fine grid). Thus only 9-digit accuracy can currently be reached with `upsampfac=1.25`.

## Error codes

In the interfaces, the returned value is 0 if successful, otherwise the error code has the following meanings (see `src/utils.h`):

```
1  requested tolerance epsilon too small
2  attemped to allocate internal arrays larger than MAX_NF (defined in common.h)
3  spreader: fine grid too small
4  spreader: if chkbnds=1, a nonuniform point out of input range [-3pi,3pi]^d
5  spreader: array allocation error
6  spreader: illegal direction (should be 1 or 2)
7  upsampfac too small (should be >1)
8  upsampfac not a value with known Horner eval: currently 2.0 or 1.25 only
9  ndata not valid (should be >= 1)
```

In the interfaces below, `int64` (typedefed as `BIGINT` in the code) means 64-bit signed integer type, ie `int64_t`. This is used for all potentially large integers, in case the user wants large problems involving more than 2^31 points. `int` is the usual 32-bit signed integer. The `FLT` type is, as above, either `double` or `single`.

## 1D transforms

```
int finufft1d1(int64 nj,double* xj,dcomplex* cj,int iflag,double eps,int64 ms,
               dcomplex* fk, nufft_opts opts)

Type-1 1D complex nonuniform FFT.

               nj-1
   fk(k1) =    SUM cj[j] exp(+/-i k1 xj(j))  for -ms/2 <= k1 <= (ms-1)/2
               j=0
Inputs:
   nj     number of sources (int64)
   xj     location of sources (size-nj FLT array), in [-3pi,3pi]
   cj     size-nj FLT complex array of source strengths
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms     number of Fourier modes computed, may be even or odd (int64);
          in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-ms FLT complex array of Fourier transform values
          stored as alternating Re & Im parts (2*ms FLTs)
          order determined by opts.modeord.
   returned value - 0 if success, else see ../docs/usage.rst

   The type 1 NUFFT proceeds in three main steps (see [GL]):
   1) spread data to oversampled regular mesh using kernel.
   2) compute FFT on uniform mesh
   3) deconvolve by division of each Fourier mode independently by the kernel
      Fourier series coeffs (not merely FFT of kernel), shuffle to output.

 Written with FFTW style complex arrays. Step 3a internally uses dcomplex,
 and Step 3b internally uses real arithmetic and FFTW style complex.
 Because of the former, compile with -Ofast in GNU.
```

```
int finufft1d2(int64 nj,double* xj,dcomplex* cj,int iflag,double eps,int64 ms,
               dcomplex* fk, nufft_opts opts)

Type-2 1D complex nonuniform FFT.

   cj[j] = SUM   fk[k1] exp(+/-i k1 xj[j])      for j = 0,...,nj-1
           k1
   where sum is over -ms/2 <= k1 <= (ms-1)/2.

 Inputs:
   nj     number of targets (int64)
   xj     location of targets (size-nj FLT array), in [-3pi,3pi]
   fk     complex Fourier transform values (size ms, ordering set by opts.modeord)
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int).
   eps    precision requested (>1e-16)
   ms     number of Fourier modes input, may be even or odd (int64);
          in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   cj     complex FLT array of nj answers at targets
   returned value - 0 if success, else see ../docs/usage.rst

   The type 2 algorithm proceeds in three main steps (see [GL]).
   1) deconvolve (amplify) each Fourier mode, dividing by kernel Fourier coeff
   2) compute inverse FFT on uniform fine grid
   3) spread (dir=2, ie interpolate) data to regular mesh
   The kernel coeffs are precomputed in what is called step 0 in the code.

 Written with FFTW style complex arrays. Step 0 internally uses dcomplex,
 and Step 1 internally uses real arithmetic and FFTW style complex.
 Because of the former, compile with -Ofast in GNU.



int finufft1d3(int64 nj,double* xj,dcomplex* cj,int iflag, double eps,
               int64 nk, double* s, dcomplex* fk, nufft_opts opts)

Type-3 1D complex nonuniform FFT.

             nj-1
   fk[k]  =  SUM   c[j] exp(+-i s[k] xj[j]),      for k = 0, ..., nk-1
             j=0
 Inputs:
   nj     number of sources (int64)
   xj     location of sources on real line (nj-size array of FLT)
   cj     size-nj FLT complex array of source strengths
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   nk     number of frequency target points (int64)
   s      frequency locations of targets in R.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-nk FLT complex Fourier transform values at target
          frequencies sk
   returned value - 0 if success, else see ../docs/usage.rst
```

```
   The type 3 algorithm is basically a type 2 (which is implemented precisely
   as call to type 2) replacing the middle FFT (Step 2) of a type 1. See [LG].
   Beyond this, the new twists are:
   i) nf1, number of upsampled points for the type-1, depends on the product
     of interval widths containing input and output points (X*S).
   ii) The deconvolve (post-amplify) step is division by the Fourier transform
     of the scaled kernel, evaluated on the *nonuniform* output frequency
     grid; this is done by direct approximation of the Fourier integral
     using quadrature of the kernel function times exponentials.
   iii) Shifts in x (real) and s (Fourier) are done to minimize the interval
     half-widths X and S, hence nf1.
```

## 2D transforms

```
int finufft2d1(int64 nj,double* xj,double *yj,dcomplex* cj,int iflag,
               double eps, int64 ms, int64 mt, dcomplex* fk, nufft_opts opts)

Type-1 2D complex nonuniform FFT.

                   nj-1
   f[k1,k2] =      SUM  c[j] exp(+-i (k1 x[j] + k2 y[j]))
                   j=0

   for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2.

   The output array is k1 (fast), then k2 (slow), with each dimension
   determined by opts.modeord.
   If iflag>0 the + sign is used, otherwise the - sign is used,
   in the exponential.

 Inputs:
   nj     number of sources (int64)
   xj,yj     x,y locations of sources (each a size-nj FLT array) in [-3pi,3pi]
   cj     size-nj complex FLT array of source strengths,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms,mt  number of Fourier modes requested in x and y (int64);
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     complex FLT array of Fourier transform values
          (size ms*mt, fast in ms then slow in mt,
          ie Fortran ordering).
   returned value - 0 if success, else see ../docs/usage.rst

   The type 1 NUFFT proceeds in three main steps (see [GL]):
   1) spread data to oversampled regular mesh using kernel.
   2) compute FFT on uniform mesh
   3) deconvolve by division of each Fourier mode independently by the
      Fourier series coefficient of the kernel.
   The kernel coeffs are precomputed in what is called step 0 in the code.



int finufft2d2(int64 nj,double* xj,double *yj,dcomplex* cj,int iflag,double eps,
```

```
             int64 ms, int64 mt, dcomplex* fk, nufft_opts opts)

 Type-2 2D complex nonuniform FFT.

   cj[j] =  SUM   fk[k1,k2] exp(+/-i (k1 xj[j] + k2 yj[j]))      for j = 0,...,nj-1
           k1,k2
   where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,

  Inputs:
   nj     number of targets (int64)
   xj,yj     x,y locations of targets (each a size-nj FLT array) in [-3pi,3pi]
   fk     FLT complex array of Fourier transform values (size ms*mt,
          changing fast in ms then slow in mt, as in Fortran)
          Along each dimension the ordering is set by opts.modeord.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   ms,mt  numbers of Fourier modes given in x and y (int64)
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2].
   opts   struct controlling options (see finufft.h)
 Outputs:
   cj     size-nj complex FLT array of target values
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   returned value - 0 if success, else see ../docs/usage.rst

   The type 2 algorithm proceeds in three main steps (see [GL]).
   1) deconvolve (amplify) each Fourier mode, dividing by kernel Fourier coeff
   2) compute inverse FFT on uniform fine grid
   3) spread (dir=2, ie interpolate) data to regular mesh
   The kernel coeffs are precomputed in what is called step 0 in the code.



int finufft2d3(int64 nj,double* xj,double* yj,dcomplex* cj,int iflag,
    double eps, int64 nk, double* s, double *t, dcomplex* fk, nufft_opts opts)

 Type-3 2D complex nonuniform FFT.

             nj-1
   fk[k]  =  SUM   c[j] exp(+-i (s[k] xj[j] + t[k] yj[j]),    for k=0,...,nk-1
             j=0
 Inputs:
   nj     number of sources (int64)
   xj,yj  x,y location of sources in the plane R^2 (each size-nj FLT array)
   cj     size-nj complex FLT array of source strengths,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   nk     number of frequency target points (int64)
   s,t    (k_x,k_y) frequency locations of targets in R^2.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (>1e-16)
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     size-nk complex FLT Fourier transform values at the
          target frequencies sk
   returned value - 0 if success, else see ../docs/usage.rst

   The type 3 algorithm is basically a type 2 (which is implemented precisely
   as call to type 2) replacing the middle FFT (Step 2) of a type 1. See [LG].
```

```
   Beyond this, the new twists are:
   i) number of upsampled points for the type-1 in each dim, depends on the
     product of interval widths containing input and output points (X*S), for
     that dim.
   ii) The deconvolve (post-amplify) step is division by the Fourier transform
     of the scaled kernel, evaluated on the *nonuniform* output frequency
     grid; this is done by direct approximation of the Fourier integral
     using quadrature of the kernel function times exponentials.
   iii) Shifts in x (real) and s (Fourier) are done to minimize the interval
     half-widths X and S, hence nf, in each dim.
```

## 3D transforms

```
int finufft3d1(int64 nj,double* xj,double *yj,double *zj,dcomplex* cj,int iflag,
               double eps, int64 ms, int64 mt, int64 mu, dcomplex* fk,
               nufft_opts opts)

 Type-1 3D complex nonuniform FFT.

                      nj-1
   f[k1,k2,k3] =      SUM  c[j] exp(+-i (k1 x[j] + k2 y[j] + k3 z[j]))
                      j=0

      for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2,
          -mu/2 <= k3 <= (mu-1)/2.

    The output array is as in opt.modeord in each dimension.
   k1 changes is fastest, k2 middle,
   and k3 slowest, ie Fortran ordering. If iflag>0 the + sign is
   used, otherwise the - sign is used, in the exponential.

 Inputs:
   nj     number of sources (int64)
   xj,yj,zj   x,y,z locations of sources (each size-nj FLT array) in [-3pi,3pi]
   cj     size-nj complex FLT array of source strengths,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested
   ms,mt,mu  number of Fourier modes requested in x,y,z (int64);
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
   opts   struct controlling options (see finufft.h)
 Outputs:
   fk     complex FLT array of Fourier transform values (size ms*mt*mu,
          changing fast in ms to slowest in mu, ie Fortran ordering).
   returned value - 0 if success, else see ../docs/usage.rst

   The type 1 NUFFT proceeds in three main steps (see [GL]):
   1) spread data to oversampled regular mesh using kernel.
   2) compute FFT on uniform mesh
   3) deconvolve by division of each Fourier mode independently by the
      Fourier series coefficient of the kernel.
   The kernel coeffs are precomputed in what is called step 0 in the code.



int finufft3d2(int64 nj,double* xj,double *yj,double *zj,dcomplex* cj,
```

```
             int iflag,double eps, int64 ms, int64 mt, int64 mu,
             dcomplex* fk, nufft_opts opts)

 Type-2 3D complex nonuniform FFT.

   cj[j] =    SUM   fk[k1,k2,k3] exp(+/-i (k1 xj[j] + k2 yj[j] + k3 zj[j]))
           k1,k2,k3
    for j = 0,...,nj-1
   where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,
                     -mu/2 <= k3 <= (mu-1)/2

 Inputs:
   nj     number of targets (int64)
   xj,yj,zj  x,y,z locations of targets (each size-nj FLT array) in [-3pi,3pi]
   fk     FLT complex array of Fourier series values (size ms*mt*mu,
          changing fastest in ms to slowest in mu, ie Fortran ordering).
          (ie, stored as alternating Re & Im parts, 2*ms*mt*mu FLTs)
          Along each dimension, opts.modeord sets the ordering.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested
   ms,mt,mu  numbers of Fourier modes given in x,y,z (int64);
          each may be even or odd;
          in either case the mode range is integers lying in [-m/2, (m-1)/2].
   opts   struct controlling options (see finufft.h)
 Outputs:
   cj     size-nj complex FLT array of target values,
          (ie, stored as 2*nj FLTs interleaving Re, Im).
   returned value - 0 if success, else see ../docs/usage.rst

   The type 2 algorithm proceeds in three main steps (see [GL]).
   1) deconvolve (amplify) each Fourier mode, dividing by kernel Fourier coeff
   2) compute inverse FFT on uniform fine grid
   3) spread (dir=2, ie interpolate) data to regular mesh
   The kernel coeffs are precomputed in what is called step 0 in the code.



int finufft3d3(int64 nj,double* xj,double* yj,double *zj, dcomplex* cj,
             int iflag, double eps, int64 nk, double* s, double *t,
             double *u, dcomplex* fk, nufft_opts opts)

 Type-3 3D complex nonuniform FFT.

             nj-1
   fk[k]  =  SUM   c[j] exp(+-i (s[k] xj[j] + t[k] yj[j] + u[k] zj[j]),
             j=0

Inputs:
   nj     number of sources (int64)
   xj,yj,zj     x,y,z location of sources in R^3 (each size-nj FLT array)
   cj     size-nj complex FLT array of source strengths
          (ie, interleaving Re & Im parts)
   nk     number of frequency target points (int64)
   s,t,u      (k_x,k_y,k_z) frequency locations of targets in R^3.
   iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
   eps    precision requested (FLT)
   opts   struct controlling options (see finufft.h)
 Outputs:
```

```
   fk     size-nk complex FLT array of Fourier transform values at the
          target frequencies sk
   returned value - 0 if success, else see ../docs/usage.rst
                       for k=0,...,nk-1

   The type 3 algorithm is basically a type 2 (which is implemented precisely
   as call to type 2) replacing the middle FFT (Step 2) of a type 1. See [LG].
   Beyond this, the new twists are:
   i) number of upsampled points for the type-1 in each dim, depends on the
     product of interval widths containing input and output points (X*S), for
     that dim.
   ii) The deconvolve (post-amplify) step is division by the Fourier transform
     of the scaled kernel, evaluated on the *nonuniform* output frequency
     grid; this is done by direct approximation of the Fourier integral
     using quadrature of the kernel function times exponentials.
   iii) Shifts in x (real) and s (Fourier) are done to minimize the interval
     half-widths X and S, hence nf, in each dim.
```

## Interfaces from C

The C user should initialize the options struct via:

```
nufft_c_opts opts; finufft_default_c_opts(opts);
```

Options fields may then be changed in `opts` before passing to the following interfaces. We use the C99 complex type `_Complex`, which is the same as `complex`. As above, `FLT` indicates `double` or `float`. The meaning of arguments are identical to the C++ documentation above. For a demo see `examples/example1d1c.c`:

```
int finufft1d1_c(int nj,FLT* xj,FLT _Complex* cj,int iflag, FLT eps,int ms, FLT _Complex* fk, nufft_
int finufft1d2_c(int nj,FLT* xj,FLT _Complex* cj,int iflag, FLT eps,int ms, FLT _Complex* fk, nufft_
int finufft1d3_c(int j,FLT* x,FLT _Complex* c,int iflag,FLT eps,int nk, FLT* s, FLT _Complex* f, nuf
int finufft2d1_c(int nj,FLT* xj,FLT *yj,FLT _Complex* cj,int iflag, FLT eps,int ms, int mt,FLT _Compl
int finufft2d1many_c(int ndata,int nj,FLT* xj,FLT *yj,FLT _Complex* cj,int iflag, FLT eps,int ms, int
int finufft2d2_c(int nj,FLT* xj,FLT *yj,FLT _Complex* cj,int iflag, FLT eps,int ms, int mt, FLT _Comp
int finufft2d2many_c(int ndata,int nj,FLT* xj,FLT *yj,FLT _Complex* cj,int iflag, FLT eps,int ms, int
int finufft2d3_c(int nj,FLT* x,FLT *y,FLT _Complex* c,int iflag,FLT eps,int nk, FLT* s, FLT *t,FLT _C
int finufft3d1_c(int nj,FLT* xj,FLT* yj,FLT *zj,FLT _Complex* cj,int iflag, FLT eps,int ms, int mt, i
int finufft3d2_c(int nj,FLT* xj,FLT *yj,FLT *zj,FLT _Complex* cj,int iflag, FLT eps,int ms, int mt, i
int finufft3d3_c(int nj,FLT* x,FLT *y,FLT *z,FLT _Complex* c,int iflag,FLT eps,int nk, FLT* s, FLT *t
```

## Interfaces from fortran

We have not yet included control of the options in the fortran wrappers. Please help create these if you can. The meaning of arguments is as in the C++ documentation above, apart from that now `ier` is an argument which is output to. Examples of calling all 9 routines from fortran are in `fortran/nufft?d_demo.f` (for double-precision) and `fortran/nufft?d_demof.f` (single-precision). Here are the calling commands with fortran types for the default double-precision case:

```
integer ier,iflag,ms,mt,mu,nj,ndata
real*8, allocatable :: xj(:),yj(:),zj(:), sk(:),tk(:),uk(:)
real*8 err,eps
complex*16, allocatable :: cj(:), fk(:)
```

```
call finufft1d1_f(nj,xj,cj,iflag,eps, ms,fk,ier)
call finufft1d2_f(nj,xj,cj,iflag, eps, ms,fk,ier)
call finufft1d3_f(nj,xj,cj,iflag,eps, ms,sk,fk,ier)
call finufft2d1_f(nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d1many_f(ndata,nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d2_f(nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d2many_f(ndata,nj,xj,yj,cj,iflag,eps,ms,mt,fk,ier)
call finufft2d3_f(nj,xj,yj,cj,iflag,eps,nk,sk,tk,fk,ier)
call finufft3d1_f(nj,xj,yj,zj,cj,iflag,eps,ms,mt,mu,fk,ier)
call finufft3d2_f(nj,xj,yj,zj,cj,iflag,eps,ms,mt,mu,fk,ier)
call finufft3d3_f(nj,xj,yj,zj,cj,iflag,eps,nk,sk,tk,uk,fk,ier)
```

## Design notes and data types

We strongly recommend you use `upsampfac=1.25` for type-3; it reduces its run-time from around 8 times the types 1 or 2, to around 3-4 times. It is often also faster for type-1 and type-2, at low precisions.

When you include the header `finufft.h` you have access to the `BIGINT` type which is used for all potentially-large input integers (M, N, etc), and currently typedefed to `int64_t` (see `utils.h`). This allows the number of sources, number of modes, etc, to safely exceed 2^31 (around 2e9). In case you were to want to change this type, you may want to use `BIGINT` in your calling codes. Using `int64_t` will be fine if you don't change this. To change (perhaps for speed, but we have not noticed any speed hit using 64-bit integers throughout), one would change `BIGINT` from `int64_t` to `int` in `utils.h`.

Sizes >=2^31 have been tested for C++ drivers (`test/finufft?d_test.cpp`), and work fine, if you have enough RAM.

In fortran and C the interface is still 32-bit integers, limiting to array sizes <2^31.

C++ is used for all main libraries, almost entirely avoiding object-oriented code. C++ `std::complex<double>` (aliased to `dcomplex`) and FFTW complex types are mixed within the library, since to some extent it is a glorified driver for FFTW. The interfaces are dcomplex. FFTW was considered universal and essential enough to be a dependency for the whole package.

There is a hard-defined limit of `1e11` for internal FFT arrays, set in `common.h` as `MAX_NF`: if your machine has RAM of order 1TB, and you need it, set this larger and recompile. The point of this is to catch ridiculous-sized mallocs and exit gracefully. Note that mallocs smaller than this, but which still exceed available RAM, cause segfaults as usual. For simplicity of code, we do not do error checking on every malloc.

As a spreading kernel function, we use a new faster simplification of the Kaiser–Bessel kernel. At high requested precisions, like the Kaiser–Bessel, this achieves roughly half the kernel width achievable by a truncated Gaussian. Our kernel is exp(-beta.sqrt(1-(2x/W)^2)), where W = nspread is the full kernel width in grid units. This (and Kaiser–Bessel) are good approximations to the prolate spheroidal wavefunction of order zero (PSWF), being the functions of given support [-W/2,W/2] whose Fourier transform has minimal L2 norm outside of a symmetric interval. The PSWF frequency parameter (see [ORZ]) is c = pi.(1-1/2sigma).W where sigma is the upsampling parameter. See our forthcoming paper.

CHAPTER

FIVE

# ADVANCED INTERFACES FOR MANY VECTORS WITH SAME NONUNIFORM POINTS

It is common to need repeated NUFFTs with a fixed set of nonuniform points, but different strength or mode coefficient vectors. For large problems, performing sequential plain calls is efficient (although there would be a slight benefit to sorting only once), but when the problem size is smaller, certain start-up costs cause repeated calls to the plain interface to be slower than necessary. In particular, we note that FFTW takes around 0.1 ms per thread to look up stored wisdom, which for small problems (of order 10000 or less input and output data) can, sadly, dominate the runtime. Thus we include interfaces, described here, for multiple stacked strength or coefficient vectors with the same nonuniform points.

These have only been implemented for the 2d1 and 2d2 types so far, for which there are applications in cryo-EM.

## 2D transforms

```
int finufft2d1many(int ndata, BIGINT nj, FLT* xj, FLT *yj, CPX* c, int iflag,
                   FLT eps, BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

Type-1 2D complex nonuniform FFT for multiple strength vectors, same NU pts.

                  nj
  f[k1,k2,d] =   SUM  c[j,d] exp(+-i (k1 x[j] + k2 y[j]))
                 j=1

  for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, d = 0,...,ndata-1

  The output array is in increasing k1 ordering (fast), then increasing
  k2 ordering (slow), then increasing d (slowest). If iflag>0 the + sign
  is used, otherwise the - sign is used, in the exponential.
Inputs:
  ndata  number of data
  nj     number of sources (int64)
  xj,yj  x,y locations of sources (each a size-nj FLT array) in [-3pi,3pi]
  c      a size nj*ndata complex FLT array of source strengths,
         increasing fast in nj then slow in ndata.
  iflag  if >=0, uses + sign in exponential, otherwise - sign.
  eps    precision requested (>1e-16)
  ms,mt  number of Fourier modes requested in x and y; each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2].
         ms*mt must not exceed 2^31.
   opts  struct controlling options (see finufft.h)
Outputs:
  fk     complex FLT array of Fourier transform values
         (size ms*mt*ndata, increasing fast in ms then slow in mt then in ndata
         ie Fortran ordering).
```

```
  returned value - 0 if success, else see ../docs/usage.rst

  Note: nthreads times the RAM is needed, so this is good only for small problems.



int finufft2d2many(int ndata, BIGINT nj, FLT* xj, FLT *yj, CPX* c, int iflag,
                   FLT eps, BIGINT ms, BIGINT mt, CPX* fk, nufft_opts opts)

Type-2 2D complex nonuniform FFT for multiple coeff vectors, same NU pts.

  cj[j,d] =  SUM   fk[k1,k2,d] exp(+/-i (k1 xj[j] + k2 yj[j]))
           k1,k2
  for j = 0,...,nj-1,  d = 0,...,ndata-1
  where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2

Inputs:
  ndata  number of mode coefficient vectors
  nj     number of targets (int64)
  xj,yj  x,y locations of targets (each a size-nj FLT array) in [-3pi,3pi]
  fk     FLT complex array of Fourier transform values (size ms*mt*ndata,
         increasing fast in ms then slow in mt then in ndata, ie Fortran
         ordering). Along each dimension the ordering is set by opts.modeord.
  iflag  if >=0, uses + sign in exponential, otherwise - sign (int)
  eps    precision requested (>1e-16)
  ms,mt  numbers of Fourier modes given in x and y
         each may be even or odd;
         in either case the mode range is integers lying in [-m/2, (m-1)/2].
         ms*mt must not exceed 2^31.
  opts   struct controlling options (see finufft.h)
Outputs:
  cj     size-nj*ndata complex FLT array of target values, (ie, stored as
         2*nj*ndata FLTs interleaving Re, Im), increasing fast in nj then
         slow in ndata.
  returned value - 0 if success, else see ../docs/usage.rst

  Note: nthreads times the RAM is needed, so this is good only for small problems.
```

## Design notes

After extensive timing tests, we settled on blocking up the ndata vectors into blocks of size nthreads (the available thread number). Each block is handled together via FFTW and OpenMP parallelism. For instance, for type-1:

1. Each thread calls a single-threaded spreader, reusing a precomputed sorted index list.
2. Apply FFT on nthreads vectors of data using FFTW's "many dft" interface.
3. Each thread calls a single-threaded deconvolve function.

This requires ndata times the RAM overhead than the plain interface.

It would also be possible to call multi-threaded spreading, sequentially on each data vector; we found this slower in all cases, and so close to repeated calls to the plain interface as to not be useful.

For repeated small problems where the nonuniform points and strengths or coefficients change, but the mode grid is fixed, reusing the FFTW plan may still be beneficial; this would require a three-call "plan, execute, destroy" interface which we have not considered worth building yet.

CHAPTER

SIX

# MATLAB/OCTAVE INTERFACES

```
---------------------------------------------------------------------
FINUFFT1D1

[f ier] = finufft1d1(x,c,isign,eps,ms)
[f ier] = finufft1d1(x,c,isign,eps,ms,opts)

Type-1 1D complex nonuniform FFT.

                nj
    f(k1) =  SUM c[j] exp(+/-i k1 x(j))  for -ms/2 <= k1 <= (ms-1)/2
             j=1
  Inputs:
    x     location of sources on interval [-3pi,3pi], length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms     number of Fourier modes computed, may be even or odd;
           in either case the mode range is integers lying in [-ms/2, (ms-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-ms double complex array of Fourier transform values
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
---------------------------------------------------------------------------
FINUFFT1D2

[c ier] = finufft1d2(x,isign,eps,f)
[c ier] = finufft1d2(x,isign,eps,f,opts)

Type-2 1D complex nonuniform FFT.

   c[j] = SUM   f[k1] exp(+/-i k1 x[j])      for j = 1,...,nj
           k1
    where sum is over -ms/2 <= k1 <= (ms-1)/2.

 Inputs:
    x     location of NU targets on interval [-3pi,3pi], length nj
```

```
    f     complex Fourier transform values
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at targets
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
c = complex(zeros(nj,1));   % todo: change all output to inout & prealloc...
------------------------------------------------------------------------
FINUFFT1D3

[f ier] = finufft1d3(x,c,isign,eps,s)
[f ier] = finufft1d3(x,c,isign,eps,s,opts)

             nj
    f[k]  =  SUM   c[j] exp(+-i s[k] x[j]),      for k = 1, ..., nk
             j=1
  Inputs:
    x      location of NU sources in R (real line).
    c      size-nj double complex array of source strengths
    s      frequency locations of NU targets in R.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
           frequencies s
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
------------------------------------------------------------------------
FINUFFT2D1

[f ier] = finufft2d1(x,y,c,isign,eps,ms,mt)
[f ier] = finufft2d1(x,y,c,isign,eps,ms,mt,opts)

Type-1 2D complex nonuniform FFT.

                  nj
    f[k1,k2] =   SUM  c[j] exp(+-i (k1 x[j] + k2 y[j]))
                 j=1

    for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2.

  Inputs:
```

```
    x,y   locations of NU sources on the square [-3pi,3pi]^2, each length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt  number of Fourier modes requested in x & y; each may be even or odd
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms*mt) double complex array of Fourier transform values
          (ordering given by opts.modeord in each dimension, ms fast, mt slow)
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
-----------------------------------------------------------------------
FINUFFT2D1MANY

[f ier] = finufft2d1many(x,y,c,isign,eps,ms,mt)
[f ier] = finufft2d1many(x,y,c,isign,eps,ms,mt,opts)

Type-1 2D complex nonuniform FFT

                     nj
    f[k1,k2,d] =    SUM  c[j,d] exp(+-i (k1 x[j] + k2 y[j]))
                    j=1

    for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, d = 1, ..., ndata

  Inputs:
    x,y   locations of NU sources on the square [-3pi,3pi]^2, each length nj
    c     size-(nj,ndata) complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt  number of Fourier modes requested in x & y; each may be even or odd
          in either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms,mt,ndata) double complex array of Fourier transform values
          (ordering given by opts.modeord in each dimension, ms fast, mt slow)
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread

Note: nthreads copies of the fine grid are allocated, limiting this to smaller
 problem sizes.
```

```
------------------------------------------------------------------------
FINUFFT2D2

[c ier] = finufft2d2(x,y,isign,eps,f)
[c ier] = finufft2d2(x,y,isign,eps,f,opts)

Type-2 2D complex nonuniform FFT.

   c[j] =  SUM   f[k1,k2] exp(+/-i (k1 x[j] + k2 y[j]))  for j = 1,..,nj
          k1,k2
    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,

 Inputs:
    x,y   location of NU targets on the square [-3pi,3pi]^2, each length nj
    f     size (ms,mt) complex Fourier transform value matrix
          (mode ordering given by opts.modeord in each dimension)
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
------------------------------------------------------------------------
FINUFFT2D2MANY

[c ier] = finufft2d2many(x,y,isign,eps,f)
[c ier] = finufft2d2many(x,y,isign,eps,f,opts)

Type-2 2D complex nonuniform FFT.

   c[j,d] =  SUM   f[k1,k2,d] exp(+/-i (k1 x[j] + k2 y[j]))
            k1,k2
         for j = 1,..,nj, d = 1,...,ndata
    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,

 Inputs:
    x,y   location of NU targets on the square [-3pi,3pi]^2, each length nj
    f     size (ms,mt,ndata) complex Fourier transform value matrix
          (mode ordering given by opts.modeord in each dimension)
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
```

```
    c     complex double array of nj*ndata answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread

Note: nthreads copies of the fine grid are allocated, limiting this to smaller
 problem sizes.
---------------------------------------------------------------------------
FINUFFT2D3

[f ier] = finufft2d3(x,y,c,isign,eps,s,t)
[f ier] = finufft2d3(x,y,c,isign,eps,s,t,opts)

             nj
    f[k]  =  SUM   c[j] exp(+-i (s[k] x[j] + t[k] y[j])),  for k = 1, ..., nk
             j=1
  Inputs:
    x,y    location of NU sources in R^2, each length nj.
    c      size-nj double complex array of source strengths
    s,t    frequency locations of NU targets in R^2.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
           frequencies s,t
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
---------------------------------------------------------------------------
FINUFFT3D1

[f ier] = finufft3d1(x,y,z,c,isign,eps,ms,mt,mu)
[f ier] = finufft3d1(x,y,z,c,isign,eps,ms,mt,mu,opts)

Type-1 3D complex nonuniform FFT.

                      nj
    f[k1,k2,k3] =     SUM  c[j] exp(+-i (k1 x[j] + k2 y[j] + k3 z[j]))
                      j=1

    for -ms/2 <= k1 <= (ms-1)/2,  -mt/2 <= k2 <= (mt-1)/2,
        -mu/2 <= k3 <= (mu-1)/2.

  Inputs:
    x,y,z locations of NU sources on [-3pi,3pi]^3, each length nj
    c     size-nj complex array of source strengths
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps     precision requested (>1e-16)
    ms,mt,mu number of Fourier modes requested in x,y and z; each may be
          even or odd.
          In either case the mode range is integers lying in [-m/2, (m-1)/2]
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
```

```
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size (ms*mt*mu) double complex array of Fourier transform values
           (ordering given by opts.modeord in each dimension, ms fastest, mu
            slowest).
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
--------------------------------------------------------------------------
FINUFFT3D2

[c ier] = finufft3d2(x,y,z,isign,eps,f)
[c ier] = finufft3d2(x,y,z,isign,eps,f,opts)

Type-2 3D complex nonuniform FFT.

   c[j] =   SUM   f[k1,k2,k3] exp(+/-i (k1 x[j] + k2 y[j] + k3 z[j]))
          k1,k2,k3
                           for j = 1,..,nj
    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,
                      -mu/2 <= k3 <= (mu-1)/2.

 Inputs:
    x,y,z location of NU targets on cube [-3pi,3pi]^3, each length nj
    f     size (ms,mt,mu) complex Fourier transform value matrix
          (ordering given by opts.modeord in each dimension; ms fastest to mu
           slowest).
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.modeord: 0 (CMCL increasing mode ordering, default), 1 (FFT ordering)
    opts.chkbnds: 0 (don't check NU points valid), 1 (do, default).
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
 Outputs:
    c     complex double array of nj answers at the targets.
    ier - 0 if success, else:
          1 : eps too small
          2 : size of arrays to malloc exceed MAX_NF
          other codes: as returned by cnufftspread
--------------------------------------------------------------------------
FINUFFT3D3

[f ier] = finufft3d3(x,y,z,c,isign,eps,s,t,u)
[f ier] = finufft3d3(x,y,z,c,isign,eps,s,t,u,opts)

             nj
    f[k]  =  SUM   c[j] exp(+-i (s[k] x[j] + t[k] y[j] + u[k] z[j])),
             j=1
                           for k = 1, ..., nk
```

```
  Inputs:
    x,y,z  location of NU sources in R^3, each length nj.
    c      size-nj double complex array of source strengths
    s,t,u   frequency locations of NU targets in R^3.
    isign  if >=0, uses + sign in exponential, otherwise - sign.
    eps    precision requested (>1e-16)
    opts.debug: 0 (silent, default), 1 (timing breakdown), 2 (debug info).
    opts.nthreads sets requested number of threads (else automatic)
    opts.spread_sort: 0 (don't sort NU pts), 1 (do), 2 (auto, default)
    opts.fftw: 0 (use FFTW_ESTIMATE, default), 1 (use FFTW_MEASURE)
    opts.upsampfac: either 2.0 (default), or 1.25 (low RAM, smaller FFT size)
  Outputs:
    f     size-nk double complex Fourier transform values at target
          frequencies s,t,u
    returned value - 0 if success, else:
                     1 : eps too small
                     2 : size of arrays to malloc exceed MAX_NF
```

A note on integer sizes: In Matlab/MEX, mwrap uses `int` types, so that output arrays can only be <2^31. However, input arrays >=2^31 have been tested, and while they don't crash, they result in wrong answers (all zeros). This has yet to be fixed (please help; an updated version of mwrap might be needed).

For a full list of error codes see *Error codes*.

CHAPTER

SEVEN

# PYTHON INTERFACE

These python interfaces are by Daniel Foreman-Mackey, Jeremy Magland, and Alex Barnett, with help from David Stein. See the installation notes for how to install these interfaces. Below is the documentation for the nine routines. The 2d1 and 2d2 "many vector" interfaces are now also included.

Notes:

1. The module has been designed not to recompile the C++ library; rather, it links to the existing static library.
2. In the below, "float" and "complex" refer to double-precision for the default library. One can compile the library for single-precision, but the python interfaces are untested in this case.
3. NumPy input and output arrays are generally passed directly without copying, which helps efficiency in large low-accuracy problems. In 2D and 3D, copying is avoided when arrays are Fortran-ordered; hence choose this ordering in your python code if you are able (see `python_tests/accuracy_speed_tests.py`).
4. Fortran-style writing of the output to a preallocated NumPy input array is used. That is, such an array is treated as a pointer into which the output is written. This avoids creation of new arrays. The python call return value is merely a status indicator.

finufftpy.**nufft1d1**(*x*, *c*, *isign*, *eps*, *ms*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

1D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
             nj-1
    f(k1) =  SUM c[j] exp(+/-i k1 x(j))   for -ms/2 <= k1 <= (ms-1)/2
             j=0
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source points, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes requested, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **f** (*complex[ms]*) – output Fourier mode values. Should be initialized as a numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown).
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (do sort), 2 (heuristic decision to sort)

- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the f array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python_tests/demo1d1.py`

`finufftpy.`**nufft1d2** (*x*, *c*, *isign*, *eps*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

1D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] = SUM   f[k1] exp(+/-i k1 x[j])      for j = 0,...,nj-1
            k1

        where sum is over -ms/2 <= k1 <= (ms-1)/2.
```

**Parameters**

- **x** (*float[nj]*) – nonuniform target points, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms]*) – Fourier mode coefficients, where ms is even or odd In either case the mode indices are integers in [-ms/2, (ms-1)/2]
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the c array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft1d3`**(*x*, *c*, *isign*, *eps*, *s*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *upsampfac=2.0*)

1D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
            nj-1
f[k]   =    SUM   c[j] exp(+-i s[k] x[j]),      for k = 0, ..., nk-1
            j=0
```

**Parameters**

- **`x`** (*`float[nj]`*) – nonuniform source points, in R
- **`c`** (*`complex[nj]`*) – source strengths
- **`isign`** (*`int`*) – if >=0, uses + sign in exponential, otherwise - sign
- **`eps`** (*`float`*) – precision requested (>1e-16)
- **`s`** (*`float[nk]`*) – nonuniform target frequency points, in R
- **`f`** (*`complex[nk]`*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **`debug`** (*`int, optional`*) – 0 (silent), 1 (print timing breakdown)
- **`spread_debug`** (*`int, optional`*) – 0 (silent), 1, 2... (print spreader info)
- **`spread_sort`** (*`int, optional`*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **`fftw`** (*`int, optional`*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **`upsampfac`** (*`float`*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the f array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d1`**(*x*, *y*, *c*, *isign*, *eps*, *ms*, *mt*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

2D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
                nj-1
    f(k1,k2) =  SUM c[j] exp(+/-i (k1 x(j) + k2 y[j])),
                j=0
                      for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform source y-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes in x-direction, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **mt** (*int*) – number of Fourier modes in y-direction, may be even or odd; in either case the modes are integers lying in [-mt/2, (mt-1)/2]
- **f** (*complex[ms,mt]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fast, mt slow) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the f array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python/tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d1many`**(*x*, *y*, *c*, *isign*, *eps*, *ms*, *mt*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

**2D type-1 (aka adjoint) complex nonuniform fast Fourier transform, for** multiple strength vectors with same nonuniform points.

```
                  nj-1
    f(k1,k2,d) =  SUM c[j,d] exp(+/-i (k1 x(j) + k2 y[j])),
                  j=0
                      for -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2,
                      d = 0,...,ndata-1
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform source y-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj,ndata]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes in x-direction, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]
- **mt** (*int*) – number of Fourier modes in y-direction, may be even or odd; in either case the modes are integers lying in [-mt/2, (mt-1)/2]
- **f** (*complex[ms,mt,ndata]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fast, mt slow) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

---

**Note:** The outputs are written into the f array.

For small problems this routine will be faster than repeated calls to nufft2d1.

Nthreads copies of the fine grid are allocated, limiting this to smaller problem sizes than the plain 2d1 interface.

---

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python/tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d2`**(*x*, *y*, *c*, *isign*, *eps*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

2D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] =   SUM   f[k1,k2] exp(+/-i (k1 x[j] + k2 y[j])),  for j = 0,...,nj-1
           k1,k2

    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2
```

**Parameters**

- **x** (*float[nj]*) – nonuniform target x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform target y-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms,mt]*) – Fourier mode coefficients, where ms and mt are either even or odd; in either case their mode range is integers lying in [-m/2, (m-1)/2], with mode ordering in all dimensions given by modeord. Ordering is Fortran-style, ie ms fastest.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the c array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d2many`**(*x*, *y*, *c*, *isign*, *eps*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

**2D type-2 (aka forward) complex nonuniform fast Fourier transform, for** multiple coefficient vectors with same nonuniform points.

```
    c[j,d] =  SUM  f[k1,k2,d] exp(+/-i (k1 x[j] + k2 y[j])),
            k1,k2

    for j = 0,...,nj-1, and d = 0,...,ndata-1
    where sum is over -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2
```

**Parameters**

- **x** (*float[nj]*) – nonuniform target x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform target y-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj,ndata]*) – output values at targets. Should be initialized as a Fortran-ordered numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **f** (*complex[ms,mt,ndata]*) – Fourier mode coefficients, where ms and mt are either even or odd; in either case their mode range is integers lying in [-m/2, (m-1)/2], with mode ordering in all dimensions given by modeord. Ordering is Fortran-style, ie ms fastest.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

---

**Note:** The outputs are written into the c array.

For small problems this routine will be faster than repeated calls to nufft2d2.

Nthreads copies of the fine grid are allocated, limiting this to smaller problem sizes than the plain 2d2 interface.

---

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft2d3`**(*x*, *y*, *c*, *isign*, *eps*, *s*, *t*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *upsampfac=2.0*)

2D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
            nj-1
f[k]   =    SUM   c[j] exp(+-i s[k] x[j] + t[k] y[j]),  for k = 0,...,nk-1
            j=0
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source point x-coords, in R
- **y** (*float[nj]*) – nonuniform source point y-coords, in R

- **c** (*complex[nj]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **s** (*float[nk]*) – nonuniform target x-frequencies, in R
- **t** (*float[nk]*) – nonuniform target y-frequencies, in R
- **f** (*complex[nk]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the f array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d1`**(*x*, *y*, *z*, *c*, *isign*, *eps*, *ms*, *mt*, *mu*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

3D type-1 (aka adjoint) complex nonuniform fast Fourier transform

```
                  nj-1
    f(k1,k2,k3) =  SUM c[j] exp(+/-i (k1 x(j) + k2 y[j] + k3 z[j])),
                  j=0
       for -ms/2 <= k1 <= (ms-1)/2,
           -mt/2 <= k2 <= (mt-1)/2,  -mu/2 <= k3 <= (mu-1)/2
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform source y-coords, valid only in [-3pi,3pi]
- **z** (*float[nj]*) – nonuniform source z-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **ms** (*int*) – number of Fourier modes in x-direction, may be even or odd; in either case the modes are integers lying in [-ms/2, (ms-1)/2]

- **mt** (*int*) – number of Fourier modes in y-direction, may be even or odd; in either case the modes are integers lying in [-mt/2, (mt-1)/2]
- **mu** (*int*) – number of Fourier modes in z-direction, may be even or odd; in either case the modes are integers lying in [-mu/2, (mu-1)/2]
- **f** (*complex[ms,mt,mu]*) – output Fourier mode values. Should be initialized as a Fortran-ordered (ie ms fastest) numpy array of the correct size
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (prints spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the f array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.`**`nufft3d2`**(*x*, *y*, *z*, *c*, *isign*, *eps*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *modeord=0*, *chkbnds=1*, *upsampfac=2.0*)

3D type-2 (aka forward) complex nonuniform fast Fourier transform

```
    c[j] =   SUM   f[k1,k2,k3] exp(+/-i (k1 x[j] + k2 y[j] + k3 z[j])).
           k1,k2,k3
                     for j = 0,...,nj-1,  where sum is over
    -ms/2 <= k1 <= (ms-1)/2, -mt/2 <= k2 <= (mt-1)/2, -mu/2 <= k3 <= (mu-1)/2
```

**Parameters**

- **x** (*float[nj]*) – nonuniform target x-coords, valid only in [-3pi,3pi]
- **y** (*float[nj]*) – nonuniform target y-coords, valid only in [-3pi,3pi]
- **z** (*float[nj]*) – nonuniform target z-coords, valid only in [-3pi,3pi]
- **c** (*complex[nj]*) – output values at targets. Should be initialized as a numpy array of the correct size
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)

- **f** (*complex[ms,mt,mu]*) – Fourier mode coefficients, where ms, mt and mu are either even or odd; in either case their mode range is integers lying in [-m/2, (m-1)/2], with mode ordering in all dimensions given by modeord. Ordering is Fortran-style, ie ms fastest.
- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **modeord** (*int, optional*) – 0 (CMCL increasing mode ordering), 1 (FFT ordering)
- **chkbnds** (*int, optional*) – 0 (don't check NU points valid), 1 (do)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

**Note:** The output is written into the c array.

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF, 4 at least one NU point out of range (if chkbnds true)

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

`finufftpy.nufft3d3` (*x*, *y*, *z*, *c*, *isign*, *eps*, *s*, *t*, *u*, *f*, *debug=0*, *spread_debug=0*, *spread_sort=2*, *fftw=0*, *upsampfac=2.0*)

3D type-3 (NU-to-NU) complex nonuniform fast Fourier transform

```
              nj-1
    f[k]   =  SUM   c[j] exp(+-i s[k] x[j] + t[k] y[j] + u[k] z[j]),
              j=0
                                                      for k = 0,...,nk-1
```

**Parameters**

- **x** (*float[nj]*) – nonuniform source point x-coords, in R
- **y** (*float[nj]*) – nonuniform source point y-coords, in R
- **z** (*float[nj]*) – nonuniform source point z-coords, in R
- **c** (*complex[nj]*) – source strengths
- **isign** (*int*) – if >=0, uses + sign in exponential, otherwise - sign
- **eps** (*float*) – precision requested (>1e-16)
- **s** (*float[nk]*) – nonuniform target x-frequencies, in R
- **t** (*float[nk]*) – nonuniform target y-frequencies, in R
- **u** (*float[nk]*) – nonuniform target z-frequencies, in R
- **f** (*complex[nk]*) – output values at target frequencies. Should be initialized as a numpy array of the correct size

- **debug** (*int, optional*) – 0 (silent), 1 (print timing breakdown)
- **spread_debug** (*int, optional*) – 0 (silent), 1, 2... (print spreader info)
- **spread_sort** (*int, optional*) – 0 (don't sort NU pts in spreader), 1 (sort), 2 (heuristic decision to sort)
- **fftw** (*int, optional*) – 0 (use FFTW_ESTIMATE), 1 (use FFTW_MEASURE)
- **upsampfac** (*float*) – either 2.0 (default), or 1.25 (low RAM & small FFT size)

---

**Note:** The output is written into the f array.

---

**Returns** 0 if success, 1 if eps too small, 2 if size of arrays to malloc exceed MAX_NF

**Return type** int

#### Example

see `python_tests/accuracy_speed_tests.py`

CHAPTER

EIGHT

# JULIA INTERFACE

Ludvig af Klinteberg has built an interface from the julia language. This interface is found at this github repo, and is actually a secondary wrapper around our python interface, so you should make sure that the latter is working first.

CHAPTER

NINE

# RELATED PACKAGES

## Interfaces to FINUFFT from other languages

- FINUFFT.jl: a julia language wrapper by Ludvig af Klinteberg (SFU). This is actually a secondary wrapper around our python interface, so you should make sure that the latter is working first.

## Packages making use of FINUFFT

Here are some packages making use of FINUFFT (please let us know others):

- sinctransform: C++ and MATLAB codes to evaluate sums of the sinc and sinc^2 kernels between arbitrary nonuniform points in 1,2, or 3 dimensions, by Hannah Lawrence (2017 summer intern at Flatiron).

CHAPTER

TEN

# KNOWN ISSUES

One should also check the github issues for the project page, https://github.com/ahbarnett/finufft/issues

Also see notes in the `TODO` file.

## Issues with library

- When requested accuracy is `1e-14` or less, it is sometimes not possible to match this, especially when there are a large number of input and/or output points. This is believed to be unavoidable round-off error.
- Currently in Mac OSX, `make lib` fails to make the shared object library (.so).
- The timing of the first FFTW call is complicated, depending on whether FFTW_ESTIMATE (the default) or FFTW_MEASURE is used. Such issues are known, and discussed in other documentation, eg https://pythonhosted.org/poppy/fft_optimization.html We would like to find a way of pre-storing some Intel-specific FFTW plans (as MATLAB does) to avoid the large FFTW_ESTIMATE planning time.
- Currently, a single library name is used for single- and multi-threaded versions. Thus, i) you need to `make clean` before changing such make options, and ii) if you wish to maintain multiple such versions you need to move them around and maintain them yourself, eg by duplicating the directory.

## Issues with interfaces

- MATLAB, octave and python cannot exceed input or output data sizes of 2^31.
- MATLAB, octave and python interfaces do not handle single precision.
- A segfault occurs a small `fft` is done in MATLAB before the first `finufft` call in a session. We believe this due to incompatibility between the versions of FFTW used. We have fixed this by building a certain `fft` call into the MEX interface. A similar hack has been used by NFFT for the last decade. This issue does not occur with octave.

## Bug reports

If you think you have found a bug, please file an issue on the github project page, https://github.com/ahbarnett/finufft/issues Include a minimal code which reproduces the bug, along with details about your machine, operating system, compiler, and version of FINUFFT.

You may also contact Alex Barnett (`abarnett` at-sign `flatironinstitute.org`) with FINUFFT in the subject line.

CHAPTER

ELEVEN

# ACKNOWLEDGMENTS

The main code and mathematical development is by:

- Alex Barnett (Flatiron Institute)
- Jeremy Magland (Flatiron Institute)

Significant SIMD vectorization/acceleration of the spreader by:

- Ludvig af Klinteberg (SFU)

Other code contributions:

- Yu-Hsuan ("Melody") Shih - 2d1many, 2d2many interface for many vectors same points
- Leslie Greengard and June-Yub Lee - CMCL fortran drivers and test codes
- Dan Foreman-Mackey - python wrappers
- David Stein - python wrappers
- Dylan Simon - sphinx help

Testing, bug reports:

- Joakim Anden - catching memory leak, Matlab/FFTW issues, performance tests
- Hannah Lawrence - user testing and finding bugs
- Marina Spivak - fortran testing
- Hugo Strand - python bugs

Helpful discussions:

- Charlie Epstein - analysis of kernel Fourier transform sums
- Christian Muller - optimization (CMA-ES) for early kernel design
- Andras Pataki - complex number speed in C++
- Timo Heister - pass/fail numdiff testing ideas
- Zydrunas Gimbutas - explanation that NFFT uses Kaiser-Bessel backwards
- Vladimir Rokhlin - piecewise polynomial approximation on complex boxes

CHAPTER

# TWELVE

# REFERENCES

References for this software and the underlying mathematics include:

[FIN] A parallel non-uniform fast Fourier transform library based on an ‘‘exponential of semicircle” kernel. A. H. Barnett, J. F. Magland, and L. af Klinteberg. Preprint (2018).

[ORZ] Prolate Spheroidal Wave Functions of Order Zero: Mathematical Tools for Bandlimited Approximation. A. Osipov, V. Rokhlin, and H. Xiao. Springer (2013).

[KK] Chapter 7. System Analysis By Digital Computer. F. Kuo and J. F. Kaiser. Wiley (1967).

[FS] Nonuniform fast Fourier transforms using min-max interpolation. J. A. Fessler and B. P. Sutton. IEEE Trans. Sig. Proc., 51(2):560-74, (Feb. 2003)

[KKP] Using NFFT3—a software library for various nonequispaced fast Fourier transforms. J. Keiner, S. Kunis and D. Potts. Trans. Math. Software 36(4) (2009).

[F] Non-equispaced fast Fourier transforms with applications to tomography. K. Fourmont. J. Fourier Anal. Appl. 9(5) 431-450 (2003).

This code builds upon the CMCL NUFFT, and the Fortran wrappers are very similar to its interfaces. For that the following are references:

[GL] Accelerating the Nonuniform Fast Fourier Transform. L. Greengard and J.-Y. Lee. SIAM Review 46, 443 (2004).

[LG] The type 3 nonuniform FFT and its applications. J.-Y. Lee and L. Greengard. J. Comput. Phys. 206, 1 (2005).

The original NUFFT analysis using truncated Gaussians is:

[DR] Fast Fourier Transforms for Nonequispaced data. A. Dutt and V. Rokhlin. SIAM J. Sci. Comput. 14, 1368 (1993).

## N

nufft1d1() (in module finufftpy), 31
nufft1d2() (in module finufftpy), 32
nufft1d3() (in module finufftpy), 33
nufft2d1() (in module finufftpy), 33
nufft2d1many() (in module finufftpy), 34
nufft2d2() (in module finufftpy), 35
nufft2d2many() (in module finufftpy), 36
nufft2d3() (in module finufftpy), 37
nufft3d1() (in module finufftpy), 38
nufft3d2() (in module finufftpy), 39
nufft3d3() (in module finufftpy), 40